What Does Control Flow Really Look Like? Eyeballing the Cyclomatic Complexity Metric

Jurgen J. Vinju †

† Centrum Wiskunde & Informatica
Amsterdam, The Netherlands

Michael W. Godfrey † ‡

‡ David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, ON, Canada

Abstract

Assessing the understandability of source code remains an elusive yet highly desirable goal for software developers and their managers. While many metrics have been suggested and investigated empirically, the McCabe cyclomatic complexity metric (CC) — which is based on control flow complexity — seems to hold enduring fascination within both industry and the research community despite its known limitations. In this work, we introduce the ideas of Control Flow Patterns (CFPs) and Compressed Control Flow Patterns (CCFPs), which eliminate some repetitive structure from control flow graphs in order to emphasize high-entropy graphs. We examine eight well-known open source Java systems by grouping the CFPs of the methods into equivalence classes, and exploring the results. We observed several surprising outcomes: first, the number of unique CFPs is relatively low; second, CC often does not accurately reflect the intricacies of Java control flow; and third, methods with high CC often have very low entropy, suggesting that they may be relatively easy to understand. These findings challenge the widely-held belief that there is a clear-cut causal relationship between CC and understandability, and suggest that CC and similar measures need to be reconsidered as metrics for code understandability.

1 Introduction

Understandability of source code is an important quality attribute of software systems. It strongly influences the effectiveness of programmers who try to extend, modify, or fix the source code as it evolves within a changing social and technical environment. However, measuring understandability in a way that is repeatable, scientifically meaningful, and conforms to common sense is difficult, as there is no consensus within the community on a precise definition of the term.

While performing psychology-oriented experiments with real-world developers is one possible path, such experiments are expensive to perform and may be of limited generality. Instead, practitioners and researchers often rely on a definition of understandability that is based on concrete and measurable properties of the source code, such as counting lines of code (LOC), measuring the interface size and complexity (i.e., function points) [1], and counting the number of linearly independent control flow paths (i.e., McCabe's Cyclomatic Complexity) [2]. While some recent studies have suggested that many metrics correlate strongly with LOC [3] — which is trivial to measure — the cyclomatic complexity metric (CC) continues to be widely used as a measure of the likely understandability of source code. It is an integral part of many metric tool suites, both open-source as well as commercial. Consequently, it is worthwhile to investigate the reasonableness of using CC to measure understandability of source code.

1.1 The case for CC in-the-small

Cyclomatic complexity applied at the method level measures the number of linearly independent control flow paths within the method.[1] The intuition behind it is that to fully understand the flow of a method, one must understand at least all of its possible paths. So, high CC should indicate low understandability.

However, there are control flow idioms that do not contribute to a higher CC, yet may cause problems with understandability. For example, for a given procedure written in a structured style it is often possible to create a functionally equivalent procedure using GOTOs and "spaghetti" logic, yet with the same CC value; Figure 1 shows a somewhat contrived example in the C language. The procedural code is usually considered to be easier to understand — and harder to misunderstand — than its unstructured equivalent, yet the CC metric does not distinguish between them.

[1] "Linearly independent" means each path has some unique edges.

```
  i = 0;
  goto body;
loop:
  if (i == 10)
    goto done;
  i++;
body:
  print(i);
  goto loop;
done:
```

```
i = 0;
do
  print(i);
while (i++ != 10);
```

Figure 1. These two C-language snippets have the same functionality and the same CC value, yet the structured version on the right seems much simpler to understand.

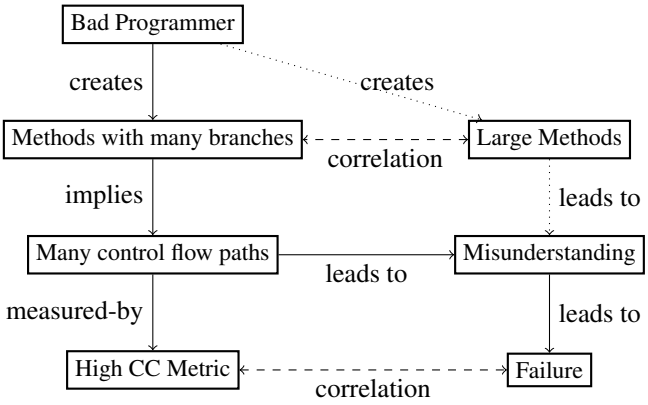

Figure 2. Two comparable explanations for correlation of high CC with failure.

At the same time, there exist control flow idioms that lead to high CC, yet would seem to be fairly easy to understand. For example, a large state machine that is implemented as a number of `switch` statements with a `case` for each outgoing edge of each state will result in high CC. Yet this design pattern seems easy to grasp conceptually, since it conforms to our mental model of a state-machine, and each of the `case` statements has the same general shape: test a condition, then activate the next state. So, a high CC value may predict low understandability where the code is in fact fairly easy to understand; that is, CC may have false positives.

1.2 The case for CC in-the-large

Cyclomatic complexity can also be applied at the system level by aggregating the values of its components; the intuition here is that systems that have many methods with high CC generally exhibit more bugs and higher maintenance costs [4]. For example, the SIG maintainability model aggregates CC by counting the percentage of LOC that contribute to methods with a high CC (> 10) as compared to the total LOC of a system [5]. Their model is applied on a daily basis to rapidly identify the "suspect" parts of large software systems.

Although the correlation of high aggregated CC with higher-than-expected maintenance problems has intuitive appeal, they may be several underlying factors at play (Figure 2). For example, the CC metric has been shown to correlate strongly with method size [3]. So, if a large system has many methods with high CC, then these are probably also the longer methods; in turn, this may indicate an inability of the programmers to form coherent abstractions and build robust, reusable units of code. So, is it this inability for high quality design that is causing poor understandability in many different ways, or is it just the high CC values?

To the best of our knowledge, there has been no analysis yet published that isolates the CC metric from other factors concerning software understanding and explicitly addresses the influence of CC on the effectiveness of programmers while doing maintenance. This paper aims to shed light in this area by studying the varieties of code control flow patterns across a set of large open source Java systems.

1.3 Contributions

In this paper we investigate the relationship between the shape of control flow patterns observed in Java methods to their CC metric values. We introduce the notions of *abstract control flow patterns* (CFPs) and *compressed control flow patterns* (CCFPs), which allow us to produce statistical evidence that the CC metric indeed does not adequately model the likely complexity of control flow in Java methods. In particular, we make the argument that an understandability metric should discount control flow patterns that exhibit internally repetitious structures — such as large `switch` statements — or occur often within the codebase and so may be examples of well known programming idioms.

2 Case studies: CC vs. control flow patterns

The control flow graph of a method is constructed from statements such as `if`, `while`, `break`, and `return` that may break the "straight line" flow of execution; Table 1 shows a full list for the Java language. These statements define the shape of the control flow graph, each adding nodes and edges.[2]

[2]Some definitions of CC model expressions, such as logical AND and OR, that can cause different branching behaviour due to short circuit evaluation. For simplicity, we consider control flow only at the statement level.

`case`	`catch`	`do-while`
`if`	`for`	`foreach`
`while`		
`block`	`break`	`continue`
`labeled`	`return`	`switch`
`synchronized`	`throw`	`try`

Table 1. The CC of a Java method is calculated by adding one for each occurrence of each keyword in the first list. The CFC is calculated by adding one for each occurrence of each keyword in both lists.

The CC metric makes a big conceptual leap in abstracting the shape of a method. It characterizes the control flow graph as simply the sum of the fan-outs of its nodes, and in so doing it flattens the dimensionality of the graph into a single number. This flattening makes comparisons and diagnoses easier, but at the cost of reduced precision and loss of information. In this work, we seek a middle ground by reducing some of the detail of a control flow graph while retaining its essential shape; this is very much in the spirit of program dependence graphs (PDGs) as used in program slicing and token-based clone detection tools.

We start by observing that the CC metric represents only a subset of all control flow statements and their net effect on the construction of the corresponding graph. One must ask: If CC is intended to measure understandability of the control flow of a method, can it afford to ignore important semantic details, such as fall-through and disruptive jumps introduced by `break`, `continue`, `return`, and `throw` statements? These statements can significantly influence the control flow semantics of method body, yet because they do not add linearly independent paths they are ignored by the CC metric.

2.1 Computing control flow patterns

In order to study a very large number of methods we introduce the notion of a "control flow pattern". Instead of studying each method as it occurs in the wild with full AST-level information, we map the methods to a normalized format that removes inessential details. More precisely,

> A *control flow pattern* (CFP) is an abstract syntax tree of a method from which all nodes that are not one of the control flow constructs have been replaced by $\perp$, and list elements that have been reduced to $\perp$ are removed.

Below is an example Java snippet that is reduced to a CFP; note that the "`x--;`" line does not add a $\perp$.

Project	#Meth	#Pat	#Pat^{comp}	#Comp
compendium	7,736	1,271 (16%)	1,234 (15%)	455 (36%)
Tomcat70	16,018	2,211 (13%)	2,158 (13%)	931 (43%)
dsbudget	306	64 (20%)	64 (20%)	18 (28%)
xml-commons-external	3,346	91 (2%)	89 (2%)	30 (33%)
apache-ant	10,278	1,391 (13%)	1,349 (13%)	555 (41%)
bcel	3,076	286 (9%)	268 (8%)	120 (44%)
hsqldb	5,326	1,013 (19%)	969 (18%)	438 (45%)
smallsql	2,556	353 (13%)	332 (12%)	158 (47%)
Merged	48,642	5,633 (11%)	5,434 (11%)	2,455 (43%)

Table 2. Control flow pattern statistics.

```
while (x >= 0) {
  if (x % 2 == 0)
    print("even");
  x--;
}
return 1;
```

```
while (⊥) {
  if (⊥)
    ⊥

}
return ⊥;
```

2.1.1 Method

We have extracted CFPs from eight large open source Java systems: `compendium`, `Tomcat70`, `dsbudget`, `xml-commons-external`, `apache-ant`, `bcel`, `hsqldb`, and `smallsql`. First, we parsed all source code for these systems and collected the abstract syntax trees of each method using the JDT library of the Rascal meta-programming environment [6]. This library accesses the Java parser of the Eclipse JDT and produces abstract syntax trees in term format. Then we applied a tree transformation to reduce each method to its representative pattern. This transformation performs a single bottom-up pass on each tree.

When applying this transformation we construct a table that maps each original method to its reduced pattern. This table is the basis from which further metrics and statistics are computed; it also allows us to trace back from each abstract CFP to the set of all methods in the original source code that the pattern models.

We note that the transformation is surjective but not injective: each Java method has a unique corresponding CFP, but several methods may map to the same representative CFP. This is by design, of course; CFPs represent the essential structure of the underlying control flow but without the distraction of inessential details.

2.1.2 Results

Table 2 summarizes the effect of reducing methods to patterns. In our dataset, we found that about 11% of the meth-

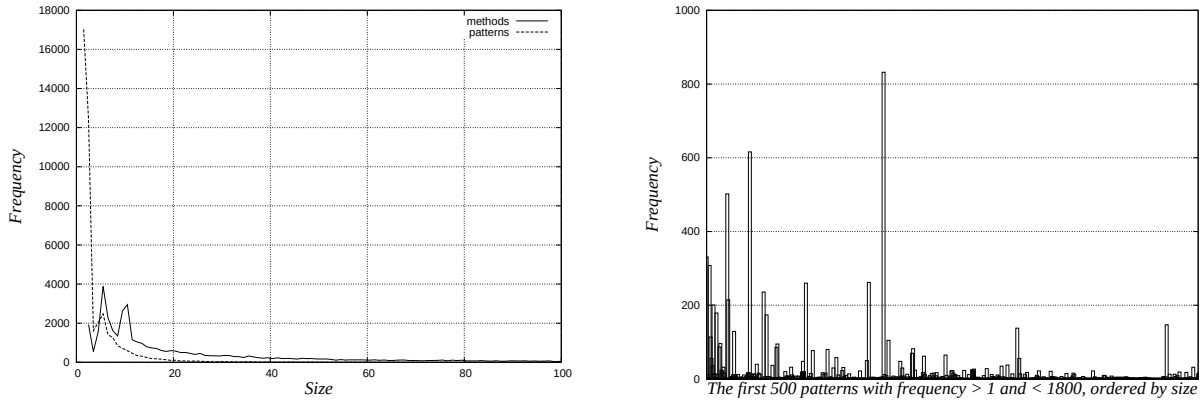

Figure 3. An overview of methods and patterns in the 8 studied systems.

ods introduced a new pattern, while the remaining 89% shared a pattern with at least one other method. Some patterns had a very high occurrence rate; for example, out of the 48,642 methods we examined, we found that three trivial patterns — "null", "calculate-an-expression-and-return-it", and "if-then-else" — had more than 1600 occurrences each.

The right-most graph in Figure 3 is a histogram that shows the frequency distribution of the patterns found in the corpus. The patterns are ordered by size from smallest to largest along the x-axis. We have removed the three most common patterns mentioned above as well as patterns that occur only once; the remaining data points in the graph are the first five hundred such patterns (and we ignore longer patterns as none of them occur very often). We can see immediately that smaller patterns occur more often than larger patterns.

2.1.3 Analysis

In theory, the number of possible CFPs increases exponentially with their size. It is in $O(n^s)$, where n is the size and s the amount of types of control flow constructs. In practice, of course, the frequency of sizes of real methods decreases rapidly: there are hardly any very long methods as compared to the very short methods.

While we have analyzed only a handful of open source Java systems, the results shown in the graphs of Figure 3 conform to common sense. Specifically, we found that:

- pattern size frequency drops off rapidly,
- method size frequency rises quickly at first, but then also drops off rapidly,
- there are many different patterns,
- there is a high degree of similarity between smaller methods, and
- pattern frequency depends heavily on pattern size.

These observations provide the background for the discussions that follow. The shape of pattern frequency distributions influences the interpretation of the following experiments: larger methods simply do not occur very often as compared to smaller patterns.

The main benefit of analyzing individual patterns as opposed to the original methods is that we can now reason about the individual patterns regardless of how many times they occur. This circumvents the effects of the observed extreme drop in frequency of the larger method bodies and allows us to focus on what CC means "in-the-small", for any given method.

2.2 What does CC miss?

To compute CC we count the occurrences of control flow branching. Yet there are many other statements that influence control flow that may need to be understood. For a start, it was recognized as early as 1984 that the CC metric ignores multiple exit points of procedures, and this can influence control flow semantics significantly [7]. Moreover, CC does not distinguish between code that uses intricate combinations of `break` and `continue` and code that uses only simple structured control flow.

2.2.1 Method

Our hypothesis is that CC does not adequately reflect all intricacies of control flow. Still, it may be the case that due to high correlation between the branching statements and

the other statements, the distinction may be of no practical consequence. Is it possible to predict with high accuracy how many other control flow statements are used, starting from the CC metric?

Lacking a "true" metric for understandability of control flow, we will use the size of the CFP as a proxy. If there is no linear correlation between this (less lossy) metric with CC, then we will argue that CC cannot meaningfully measure the understandability of a specific method either.

> The *control flow complexity* (CFC) of a CFP is the number of control flow operators (Table 1) in the original syntax tree of (any of) the methods that the pattern originates from.

The CFC of a pattern can be measured by simply counting the number of syntax tree nodes. We note that for any method m and its corresponding CFP p, the following hold:

$$CC(m) \leq CFC(m) \quad (1)$$

$$CFC(m) = CFC(p) \quad (2)$$

$$CC(m) = CC(p) \quad (3)$$

After measuring CC and CFC for all patterns we extracted from the systems under investigation, we can show distribution graphs and scatter plots. By visual inspection we can then assess if there appears to be a linear correlation.

2.2.2 Results

Figure 4 depicts the relation between the CFC and CC of the CFPs from our corpus. The top right scatter plot zooms in on the top left scatter plot to show the first 50 sizes of patterns. We see that there appears to be linear correlation. Cyclomatic complexity makes up for between 30% and 100% of the size of the pattern, which is consistent with results observed by Jbara et al. [8].

Yet, when we look more closely, the smaller patterns cover practically *all possible* cyclomatic complexities within the 30% to 100% range. For the larger methods, if we focus on each size of method in turn (imagine a vertical bar shooting upwards from any position on the x-axis), then we observe that cyclomatic complexity appears to be fairly randomly distributed in this range as well.

The distance to the least-squares linear fit is plotted in the bottom-right of Figure 4. It shows the error gets progressively worse for larger methods, as well as errors that are in the same order of magnitude as the measure itself for smaller errors.

2.2.3 Analysis

The plots in Figure 4 show how unrelated the CC metric is to CFC. CC is distributed between 30% and 100% of the control flow *for every specific size*. So, there is a rough linear relation between CFC and CC, but this is probably caused by the size of the method. Larger methods have both more control flow and more control flow splits. However, between control flow splits and control flow in general there seems no clear cut relation.

We hypothesize that a linear causal relation exists between the size of the method and a *minimum* amount of branches that will be at least used in most methods; this would explain the results of Jbara et al. [8], and at the same time account for the broad spectrum of values that we see in our results.

From the distribution patterns of CC and CFC we see that CFC is much flatter and thus has more distinctive capability then CC does. CC has a tendency of mapping larger set of methods into the same bucket than CFC does. Specifically around before threshold of 10, which is commonly used as a badness threshold for CC [5], we see that CC equates many different kinds of CFPs. From this we may stipulate that in the lower regions, CC misses the accuracy to detect problematic CFPs.

For the really large methods, with CC > 20, one could argue that the distinction is mostly irrelevant. The methods become hard to understand and high CC in this cases indicates poor understandability just as accurately as CFC or LOC do. Still, in medium-range methods, a relatively "harmless" CC count easily masks messy control flow full of `breaks`, `continues` and similar (and this can be observed in Figure 4 under the 10 bar for CC which has several entries above 20 for CFC).

From the above analysis we learn that indeed the other control flow statements, apart from the ones identified by CC, are a relevant factor in real open-source Java software systems. Of course, this analysis is not conclusive, but it seems clear that CC is not an accurate predictor for CFC within our corpus.

At least to understand the control flow of a method, one could argue that you would have to digest all of its control flow statements. If CC cannot approximate CFC, then this casts doubt on the ability of CC to predict the more difficult to define "understandability" of a method at all.

2.3 Is CC overzealous?

The previous analysis focused on finding code that may be more complex than CC suggests, but the other side of this question is equally interesting: What methods does CC label "complex" that may be quite easy to understand?

2.3.1 Method

To make our metrics more sensitive to measuring "true" understandability, it makes sense to de-emphasize patterns that

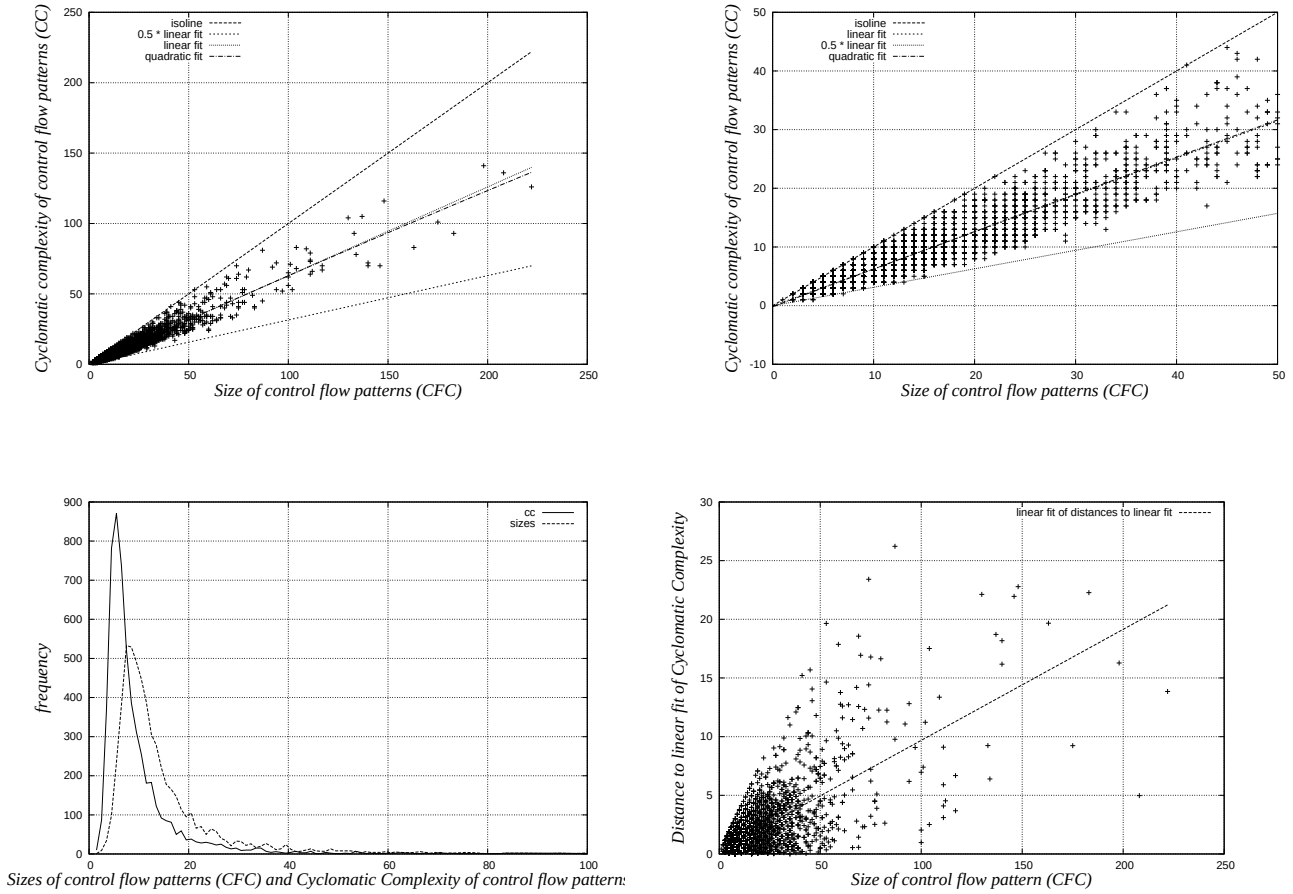

Figure 4. Comparing control flow complexity to cyclomatic complexity of control flow patterns

may score highly on CC or CFC, yet seem straightforward to comprehend. One such category are patterns that feature repetitive control structures. For example, the following pattern has a relatively high CC value of 9, but seems easy to understand due to its repetitive nature:

```
switch(⊥) {
  case ⊥ : return ⊥;
  case ⊥ : return ⊥;
  case ⊥ : return ⊥;
  case ⊥ : return ⊥;
  case ⊥ : return ⊥;
  case ⊥ : return ⊥;
  case ⊥ : return ⊥;
  case ⊥ : return ⊥;
}
```

If such regularity occurs often, then we may question the relevance of the CC metric when applied to real software systems.

We now introduce an abstraction that de-emphasizes control structures that occur repeatedly at the same structural level:

> A *compressed control flow pattern* (CCFP) is a control flow pattern where each list $e_1, e_2, \ldots, e_n$ of $n > 1$ *consecutive* tree nodes in the pattern that are structurally equal is replaced by a single node $\mathcal{R}(e_1)$.

We note that compression is performed bottom-up recursively, so that nested regularity may emerge and then be compressed again. The CCFP for the above example would be:

```
switch (⊥) {
  R (case ⊥ : return ⊥;)
}
```

We can now trivially extend the CFC metric to CCFPs:

> The *compressed control flow complexity* (CCFC) of a method is the number of nodes in its CCFP.

We note that for any method m, it must be the case that:

$$CCFC(m) \leq CFC(m) \tag{4}$$

However, we cannot automatically infer an inequality between CCFC and CC, as neither value is necessarily higher or lower than the other.

Our hypothesis is now that there should be many methods that are highly compressible. If so, then we deduce that the CC metric commonly overestimates control flow understandability. Using the above definition of a CCFP we have reduced all the patterns of the systems in Table 2. This allows us to plot the relation between the sizes of CFPs and compressed CFPs, and observe their respective distribution patterns. From this we can see how often there is repetition and how much repetition there is in the control flow of Java methods in the corpus.

2.3.2 Results

In Table 2 we see that compression occurs in more than 40% of all the patterns. At the same time, the statistics show that compression does not collapse many patterns together.

Figure 5 shows the compression per CFP; we can see that compression is common for all sizes of patterns, and that compression rates can be high for all sizes. Smaller patterns, if they compress, are more likely to compress a little than a lot. Larger patterns, of which there are many fewer, often compress highly. Using a least squares approach, we have plotted linear, quadratic, and square root fits; the data set seems to favour small compression rates (i.e., many dots are printed on top of each other). We found that the square root model fits best for this data set, confirming that compression is more effective on larger patterns.

We have also manually examined several of the larger methods within the corpus to better understand what code would compress. We found the most extreme compression occurred in code that had been automatically generated, such as by lexer and parser generators. While this may be unsurprising, it is interesting to see that such generated patterns are compressible by simply eliminating repetition.

The following is an example of a nested compressed CFP, found in the `smallsql` system:

```
switch (⊥) {
  ℛ (case ⊥: switch (⊥) {
       ℛ ( case ⊥: return ⊥; )
  })
}
```

We have simplified it here for presentation purposes by leaving out the context around the `switch` and removing some irregular cases. This pattern was associated with a single method that interprets boolean expressions in SQL with a CC of 126. The CCFC of the (real) method is 7 and the CC of the compressed pattern is 27 (the simplified pattern above has CCFC of 7 and CC of 3). The method dispatches on the types of the arguments of an expression with the outer `switch`, and then on the operator kind in the nested `switch` and computes the result of the expression via recursion. This is a straightforward design for an interpreter, and it seems easy for an experienced programmer to understand. In theory, the particular nested pattern may represent a quadratic number of methods, depending on how many repetitions occur at each level. In practice, we found that it occurs only once in the systems that we investigated.

The method with the largest compressed size (CCFC) was found in `compendium`: 179. It has an original CC of 141, its CFC is 198 and the CC of the compressed pattern is still 119. This is the worst case compression rate we found for such larger methods. The code dispatches on key press events and directly implements the associated actions. The control flow structure is governed mostly by nested if-then and if-then-else statements with an occasional nested `switch`. Since in many cases single outermost conditional span multiple pages of source text, it is difficult to see which parts of the code are mutually exclusive and which are executed in order.

The effects of compression on the distributions of sizes of patterns and their cyclomatic complexity is shown in Figure 6. Between compressed and uncompressed, the distributions have the same general shape, but compressed patterns have a larger peek below the threshold of size 10. This is a significant observation since 10 is a common threshold with CC for labeling a method to be "bad". In other words, many patterns go from being "bad" to being "good" by eliminating repetitive structure.

2.3.3 Analysis

Compressing repetition can lead to a significant reduction in pattern size. The reduction is most evident in the larger patterns, although there are comparatively fewer of them. We can conclude that compression may be used to filter large methods that are easy-to-understand patterns and perhaps even generated. However, there are so few of such larger patterns that we should not jump to the conclusion that CC is not a good way of finding hard-to-understand patterns.

For smaller patterns the compression may be less evident, yet is has significant effect on the interpretation of the metrics. Although smaller patterns are usually not compressed below 50%, the compression does affect the interpretation of the metrics via the commonly used threshold of 10. From this perspective we can learn that systems that are easy to understand because they have repetitive control flow structures may be judged harshly while they in fact have easy-to-understand control flow structures.

We conclude that CC indeed often underestimates the understandability of CFPs; it is most pronounced in larger methods (which are much less common), but it is still signif-

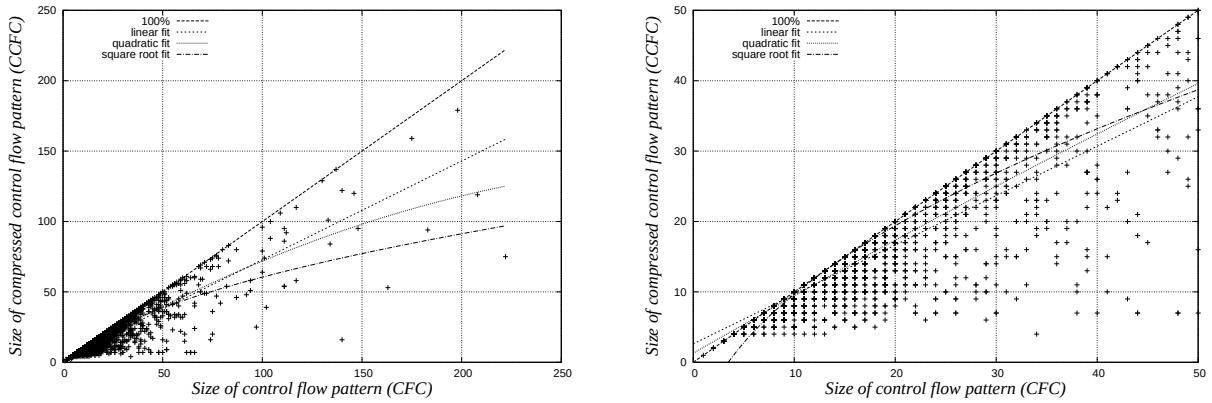

Figure 5. What compression does to the size of CFP.

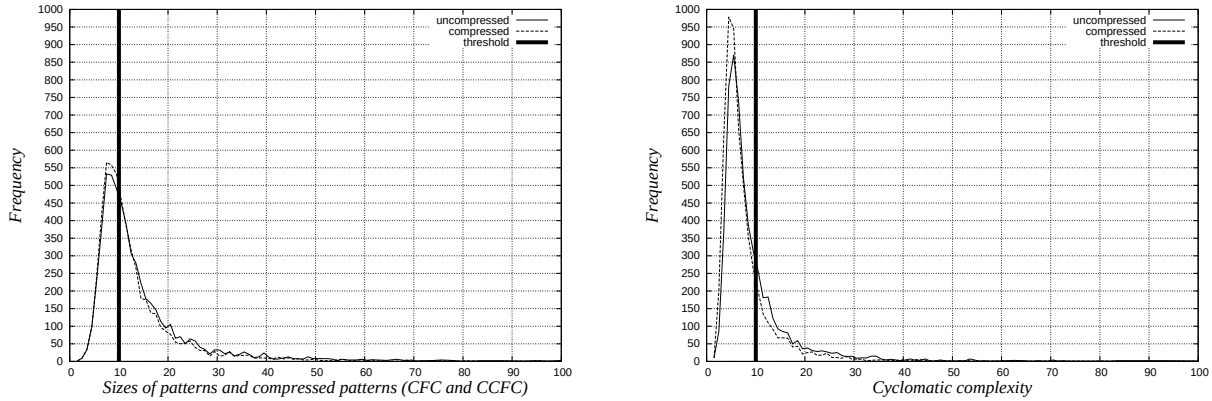

Figure 6. Size and CC distributions with and without compression

icant in the shorter methods (which are very common). This suggests that the CC metric is not very accurate for judging individual methods on understandability, and that when used for aggregation over whole systems using a threshold care must be taken in interpreting the results if there are many methods whose CFC are in the range of 10 to 20.

3 Related work

There is a large body of work on the generation, interpretation, and experimental validation of software metrics (e.g., the work by Halstead [9]). We do not have the space here to explore this topic in detail, so we mention only relevant and more recent developments.

Herraiz and Hassan argued we do not need complexity metrics because they correlate strongly with the number of lines of code (LOC) [3]. While our results also found such a correlation, we draw a slightly different conclusion. The McCabe cyclomatic complexity metric correlates in general with the size of a method because every method has at least a few branches scattered over its body. However, this does not accurately predict the complexity of the rest of the code that may or may not use more than this minimal number of branches. We distinguish explicitly between interpretation on a method-by-method basis versus a global system-to-system aggregated comparison. Herraiz and Hassan's conclusion remains valid for the latter perspective, but on the smaller scale we feel that it is reasonable to assert that there is still room for better complexity metrics.

Vasilescu et al. [10, 11] have studied the effect of different aggregations for software metrics on their interpretation, which is very noticeable. The SIG maintainability

model also pays attention directly to the effect of aggregation on their judgement of quality [5]. We have learned this lesson and avoided computing aggregate measures such as averages. Indeed, the statistical distributions shown in this paper are interpreted directly rather than framing them in a statistical model.

Jbara et al. have also investigated the relationship between CC and understandability, using the Linux kernel source code as a case study [8]. They found evidence of code that was classified as complex by CC, yet seemed to them to be "well structured" enough for understanding, and indeed was under active maintenance. Our results corroborate theirs in that respect. Jbara also concludes, like McCabe mentioned in the original paper on cyclomatic complexity [2], perhaps large `switch` cases should not be counted. We have taken this idea one step further and eliminated all locally repetitive structures. Our paper also adds another perspective, namely that cyclomatic complexity may miss the opportunity of spotting hard-to-understand code next to mislabelling easy code as complex.

Alves et al. studies the construction of benchmarks from software metrics [12]. They also notice that software metrics like CC are often distributed according to (what seems to be) a power law.[3] They automatically derive threshold values in an objective and repeatable way. It may very well be that by replacing CC with CCFM some systems fall in entire different categories in the benchmarks they produce. Our work therefore is highly relevant to the industry of software quality verification and monitoring.

4 Threats to validity

We now discuss threats to validity pertaining to the work presented here.

4.1 CC and short-circuit semantics

In the interests of conceptual simplicity, we have chosen to ignore the influence of short-circuit semantics for boolean `&&` and `||` operators in Java. Strictly speaking, these should be modelled, as they introduce additional linearly independent control flow paths into the graph; in practice, they are often ignored by existing metric suite implementations. We take the position that developers are more likely to consider understandability at the statement level than the expression level, and so we have chosen to ignore these operators; this also slightly simplifies our implementation, but this is not the primary reason for adopting it.

[3] Clauset et al. point out how hard it is to verify that a data set is accurately modelled by a power law distribution [13].

4.2 Selection of example systems

A key requirement for selecting example systems for the corpus was compatibility with our analysis front-end; that is, we selected systems that were relatively easy to compile using Eclipse. We needed fully compilable Java code because our front-end performs name and type analysis, and it will not produce a syntax tree in case of an error. All systems are open-source, and two of them are SQL interpreters, which may have an influence on our analysis.

Apart from the two databases, we chose systems from a variety of application domains. We see no specific shared characteristics that may skew the results of our analysis in favor of our conclusions. Adding more different systems to this initial experiment should confirm this.

4.3 CFC as a proxy for understandability

We used the size of CFPs to argue against the predictive capability of cyclomatic complexity for understandability. Note however, that we did not need to assume that CFC is measuring understandability in any way. Our assumption is that if CC cannot model CFC, than it certainly cannot model understandability.

This assumption may be wrong, if for example creating an accurate mental model of the control flow method does not require reading it fully. This might happen if the source code comments are of particularly high quality or if the names of the method carry a significant amount of cognitive information about the functionality. In our experience, both are relatively unlikely events: reading the source code is still the most trusted method used by developers for learning what a method does.

4.4 Do people understand control flow by recognizing patterns?

We used the occurrence of local repetitive structure in control flow patterns to search for methods that may be easier to understand than CC might indicate. The underlying assumption is that code that looks regular is easier to "chunk" [14] and therefore easier to understand. The extreme examples that were discussed are suggestive, but it may be the case that less extensive compression has less of an influence on understanding. It is interesting to observe that compression does not add to the "uniqueness" of control flow patterns much, which means that if you have seen the compressed version and understand it, you have seen an unambiguous representation of a control flow pattern that may be larger but not different.

5 Conclusions

In this paper we have investigated the relation between the shape of control flow in Java methods and the cyclomatic complexity metric (CC). We have collected empirical evidence from eight open source Java systems that suggests that CC can, and often does, underestimate and overestimate the understandability of methods. This implies that the CC metric, when applied to judge a single method on understandability, must be taken with a grain of salt. It also implies that the strategy of comparing entire systems using thresholds for high CC may have to be re-evaluated with compressibility and the presence of other control flow statements in mind.

The enabling concepts for doing these experiments are two-fold. First, we introduced control flow patterns (CFPs), an abstraction of the abstract syntax trees that removes all but control flow statements. Control flow patterns allowed us to unveil where and how many times CC underestimates the complexity of control flow. Second, we introduced compressed control flow patterns (CCFPs), which summarize all consecutively repetitive control flow structure. Compression allowed us to identify where and how many times CC might overestimate the complexity of control flow. These concepts should be applicable in a broader context of experimentally studying the shape of control flow and the relation of control flow statement usage to software maintainability.

Data availability

The corpus of eight open source Java systems, the extraction of basic data, the reduction to control flow patterns, the compression algorithm, the metrics calculation, and `csv` output files can all be found online [15].

Acknowledgements

We thank Arnoud Roo, Jasper Timmer, and Douwe Kasemier, who did some preliminary investigations into these questions as part of of their Master's work at CWI.

References

[1] "International Function Point Users Group," April 2012. [Online]. Available: http://www.ifpug.org

[2] T. McCabe, "A complexity measure," *IEEE Trans. on Software Engineering*, vol. 2, no. 4, December 1976.

[3] I. Herraiz and A. E. Hassan, "Beyond lines of code: Do we need more complexity metrics?" in *Making Software: What Really Works, and Why We Believe It*, A. Oram and G. Wilson, Eds. O'Reilly Media, 2010.

[4] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability," *IEEE Computer*, vol. 27, no. 8, August 1994.

[5] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *Proc. of* 6^{th} *Intl. Conf. on Quality of Information and Communications Technology*, September 2007.

[6] P. Klint, T. van der Storm, and J. Vinju, "RASCAL: A domain specific language for source code analysis and manipulation," in *Proc. of* 9^{th} *IEEE Intl. Working Conf. on Source Code Analysis and Manipulation*, September 2009.

[7] W. A. Harrison, "Applying McCabe's complexity measure to multiple-exit programs," *Software: Practice and Experience*, vol. 14, no. 10, October 1984.

[8] A. Jbara, A. Matan, and D. G. Feitelson, "High-MCC functions in the Linux kernel," in *Proc. of* 20^{th} *IEEE Intl. Conf. on Program Comprehension*, June 2012.

[9] M. H. Halstead, *Elements of Software Science*. New York, NY, USA: Elsevier Science Inc., 1977.

[10] B. Vasilescu, A. Serebrenik, and M. G. van den Brand, "By no means: A study on aggregating software metrics," in *Proc. of the* 2^{nd} *Intl. Workshop on Emerging Trends in Software Metrics*, May 2011.

[11] B. Vasilescu, A. Serebrenik, and M. van den Brand, "You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics," in 27^{th} *IEEE Intl. Conf. on Software Maintenance*, September 2011.

[12] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *Proc. of* 26^{th} *IEEE Intl. Conf. on Software Maintenance*, September 2010.

[13] A. Clauset, C. R. Shalizi, and M. E. J. Newman, "Power-law distributions in empirical data," *SIAM Review*, vol. 51, no. 4, June 2007.

[14] A. Von Mayrhauser and A. Vans, "Program comprehension during software maintenance and evolution," *IEEE Computer*, vol. 28, no. 8, August 1995.

[15] J. Vinju, "Experimental data and scripts." [Online]. Available: http://homepages.cwi.nl/~jurgenv/experiments/ControlFlowAnalysis/index.html